LOS LENGUAJES DE PROGRAMACIÓN

Cuando queremos comunicarnos con una persona, hacemos uso de un idioma en común para entendernos. Nosotros, con este libro, nos comunicamos por medio del lenguaje español, pero si llegara a visitarnos una persona de otro país donde no se hable español, entonces tendríamos un problema. La solución a este problema es muy sencilla: necesitamos aprender el idioma de esta persona para poder comunicarnos con ella. Lo mismo sucede con la computadora si queremos programarla, es necesario comunicarnos con ella, y la forma correcta de hacerlo es hablando su idioma. Entonces podemos decir que el idioma de la computadora se conoce como **lenguaje de programación** y el lenguaje de programación que aprenderemos es **C#**. Este lenguaje es sencillo y poderoso a su vez.
Una vez que sepamos el lenguaje de programación, entonces la computadora entenderá qué es lo que queremos que haga.
Los lenguajes de programación existen desde hace muchos años y son de diferentes tipos. Algunos han evolucionado y han surgido nuevos lenguajes como C#. Otros lenguajes son especializados para determinadas labores y muchos otros se han extinguido o han quedado obsoletos.

Los programas de computadora

Todos o casi todos sabemos cómo cocinar una torta, y de hecho, éste es un buen ejemplo para saber cómo hacer un programa de computadora. Cuando queremos cocinar una torta, lo primero que necesitamos son los ingredientes. Si falta alguno, entonces no tendremos una rica torta. Ya una vez que hemos preparado o recolectado nuestros ingredientes, seguimos la receta al pie de la letra. Ésta nos dice paso a paso qué es lo que tenemos que hacer.
Pero, ¿qué sucedería si no seguimos los pasos de la receta?, ¿qué sucede si ponemos los ingredientes primero en el horno y luego los revolvemos?, ¿el resultado es una torta? La respuesta a estas preguntas es evidente: el resultado no es una torta. De esto deducimos que para lograr algo, necesitamos una serie de pasos, pero también es necesario seguirlos en el orden adecuado. A esta serie de pasos la llamamos receta.

Un error que se puede tener al iniciarnos con la programación de computadoras con C# es olvidar colocar **;** al finalizar la sentencia. Si nuestro programa tiene algún error y todo parece estar bien, seguramente olvidamos algún punto y coma. Es bueno reconocer estos tipos de errores ya que nos permite reducir el tiempo de corrección necesario.

Cualquier persona puede pensar en una actividad y mencionar la lista de pasos y el orden en el que se deben cumplir. Podemos hacer sin problema la lista de pasos para cambiar el neumático del automóvil o para alimentar a los peces del acuario. Con esto vemos que no encontraremos problema alguno para listar los pasos de cualquier actividad que hagamos.

Ahora, seguramente nos preguntamos: ¿qué tiene esto que ver con la programación de computadoras? y la respuesta es simple. Cualquier persona que pueda hacer una lista de pasos puede programar una computadora. El programa de la computadora no es otra cosa que una lista de los pasos que la computadora tiene que seguir para hacer alguna actividad. A esta lista de pasos la llamamos **algoritmo**. Un algoritmo son los pasos necesarios para llevar a cabo una acción. Una característica importante del algoritmo es que tiene un punto de **inicio** y un punto de **fin**, lo que indica que los pasos se llevan a cabo de forma **secuencial**, uno tras otro. El algoritmo sería equivalente a la receta de la torta y los ingredientes necesarios generalmente serán los datos o la información que usaremos.

Hagamos un pequeño programa para ver cómo funcionaría esto. Para esto creamos un proyecto, tal y como vimos en el **Capítulo 1**. Nuestro programa mostrará los pasos para hacer una torta o un pastel. Colocamos el siguiente código en nuestro programa y veremos cómo se corresponde con lo que hemos aprendido.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
```

Las funciones son secciones del programa en las que podemos tener código especializado. La función **Main()** es invocada por el sistema operativo cuando queremos que nuestro programa se inicie. Es la única función que deben tener todos los programas que desarrollemos.

```
        {
            Console.WriteLine("1- Precalentar el horno");
            Console.WriteLine("2- Batir margarina con azucar");
            Console.WriteLine("3- Agregar huevos");
            Console.WriteLine("4- Cernir harina");
            Console.WriteLine("5- Agregar a la mezcla y leche");
            Console.WriteLine("6- Hornear por 40 minutos");
            Console.WriteLine("7- Decorar y comer");
        }
    }
}
```

C# es un **lenguaje orientado a objetos** y necesita que definamos una **clase** para poder crear un programa. En este momento no veremos las clases, pero regresaremos a ellas en un capítulo posterior. Lo primero que debemos encontrar es una **función** que se llama **Main()**, que es muy importante, como vimos en el **Capítulo 1.**

Ya sabemos que los algoritmos necesitan un inicio y un fin. En nuestro programa el punto donde siempre inicia es la función **Main()**. Esta función indica el punto de arranque de nuestro programa y aquí colocaremos los pasos que debe seguir nuestro programa en el orden adecuado.

Para indicar todos los pasos que pertenecen a **Main()** necesitamos colocarlos adentro de su **bloque de código**. Un bloque de código se define entre llaves **{ }**. Todo lo que se encuentre en ese bloque de código pertenece **a Main()**, lo que se encuentre afuera de él, no pertenece.

Es importante que notemos que la **clase** también tiene un bloque de código y **Main()** se encuentra adentro de él. De igual forma, el **namespace** también tiene su propio bloque de código y la clase se encuentra adentro, aunque por ahora sólo nos concentraremos en el bloque de código de **Main().**

Un error común cuando se crean bloques de código es olvidar cerrarlos. Como recomendación, una vez que abrimos un bloque de código con **{**, inmediatamente debemos de colocar el cierre con **}** y luego programar su contenido. De esta forma, no

Cuando colocamos un valor con decimales en C# automáticamente se lo interpreta como de tipo **double**. Sin embargo, si necesitamos colocar ese valor en una variable de tipo **float**, debemos usar el sufijo **f**. Éste le dice a C# que use ese valor como flotante. Ejemplo:

resultado = a * 3.57f;.

olvidaremos cerrarlo. Dentro de **Main()** encontramos los pasos de nuestro algoritmo. En este momento nos pueden parecer extraños, pero aprenderemos qué significan. Tenemos que recordar que debemos aprender el idioma de la computadora y éstas son una de sus primeras frases.

Lo que encontramos adentro de la función **Main()** es una serie de **sentencias**. Las sentencias nos permiten colocar instrucciones que nuestro programa pueda ejecutar, que son utilizadas para mostrar un mensaje en la pantalla. El mensaje que nosotros queremos mostrar en la pantalla se muestra al invocar el método **WriteLine()**. Los métodos son similares a las funciones, existen adentro de las clases y nos permiten hacer uso de las funcionalidades internas. Este método pertenece a la clase **Console**, y en esa clase encontraremos todo lo necesario para que podamos trabajar con la consola, ya sea colocando mensajes o leyendo la información proporcionada por el usuario. Como **WriteLine()** pertenece a **Console** cuando queremos utilizarlo, debemos escribir:

```
Console.WriteLine("Hola");
```

WriteLine() necesita cierta información para poder trabajar. Esta información es el mensaje que deseamos mostrar. A los métodos se les pasa la información que necesitan para trabajar por medio de sus **parámetros**. Los parámetros necesarios se colocan entre **paréntesis()**. **WriteLine()** necesitará un parámetro de tipo **cadena.**

Una **cadena** es una colección de caracteres, es decir, letras, números y signos. La cadena se delimita con **comillas dobles "**. En el ejemplo, el mensaje que queremos que se muestre es: **Hola**. Por eso pasamos el parámetro de tipo cadena **"Hola"**.

Las sentencias se finalizan con **punto y coma ;**. Esto es importante y no debemos olvidarlo ya que este carácter en particular es reconocido por C# para indicar que la sentencia escrita hasta allí, ha finalizado.

En cada una de las sentencias invocadas hasta aquí por nuestro programa, hemos enviado un mensaje tras otro con cada paso necesario para cocinar nuestro pastel o nuestra torta. Por último, sólo resta que ejecutemos el programa escrito y veamos el resultado que aparece en la pantalla.

Cuando se invoca un método es necesario no solamente colocar su nombre, sino que también se deben poner los paréntesis. Éstos se deben poner aun cuando el método no necesite parámetros. No hay que olvidar usar el paréntesis de cierre **)**. Si olvidamos colocar los paréntesis, el compilador nos marcará errores y no podremos ejecutar el programa.

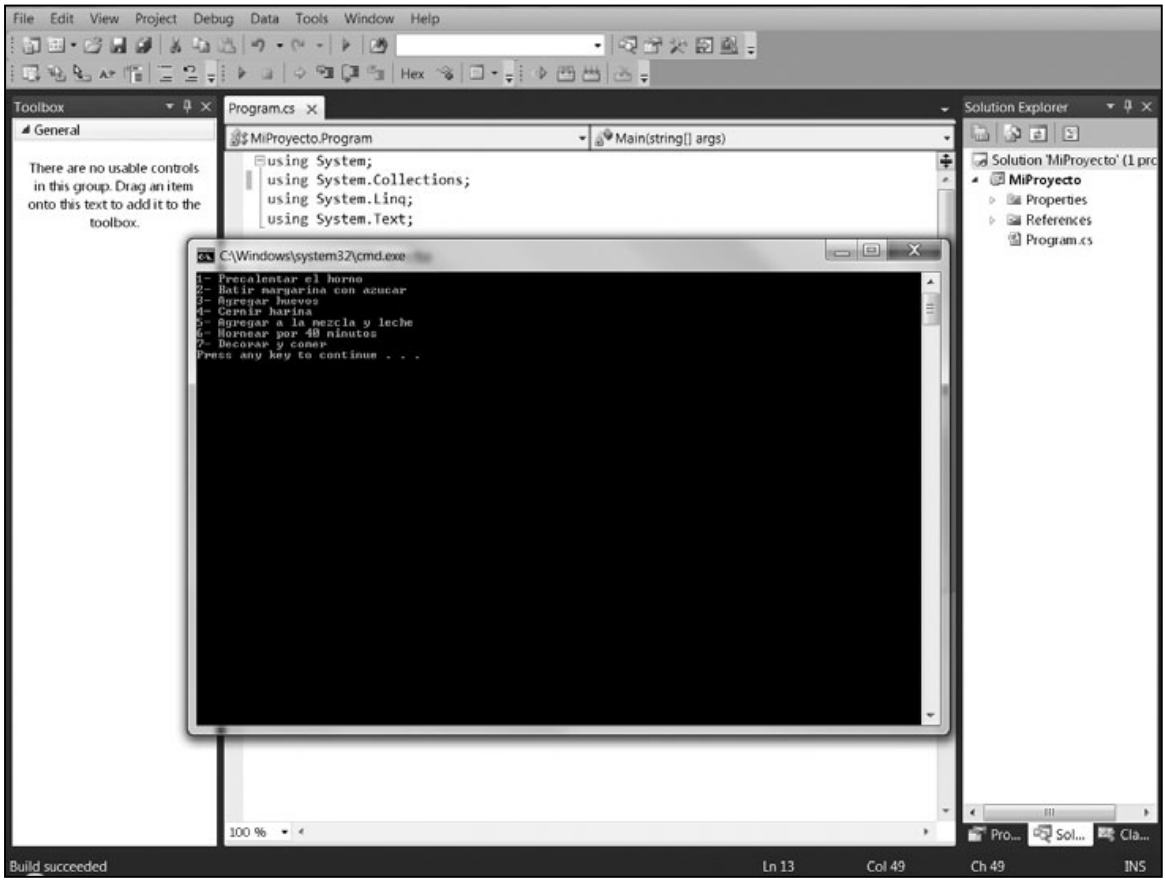

Figura 1. *Podemos observar los mensajes que enviamos a la pantalla por medio del método* ***WriteLine().***

Podemos observar que la ejecución ha mostrado los mensajes en el orden en el que los hemos puesto. Ahora podemos hacer un pequeño experimento. Cambiaremos el orden de las sentencias y observaremos qué es lo que sucede. El programa quedará de la siguiente forma:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {
            Console.WriteLine("1- Precalentar el horno");
```

```
            Console.WriteLine("2- Batir margarina con azucar");
            Console.WriteLine("6- Hornear por 40 minutos");
            Console.WriteLine("5- Agregar a la mezcla y leche");
            Console.WriteLine("7- Decorar y comer");
            Console.WriteLine("3- Agregar huevos");
            Console.WriteLine("4- Cernir harina");
        }
    }
}
```

Ejecutémoslo y veamos qué ocurre.

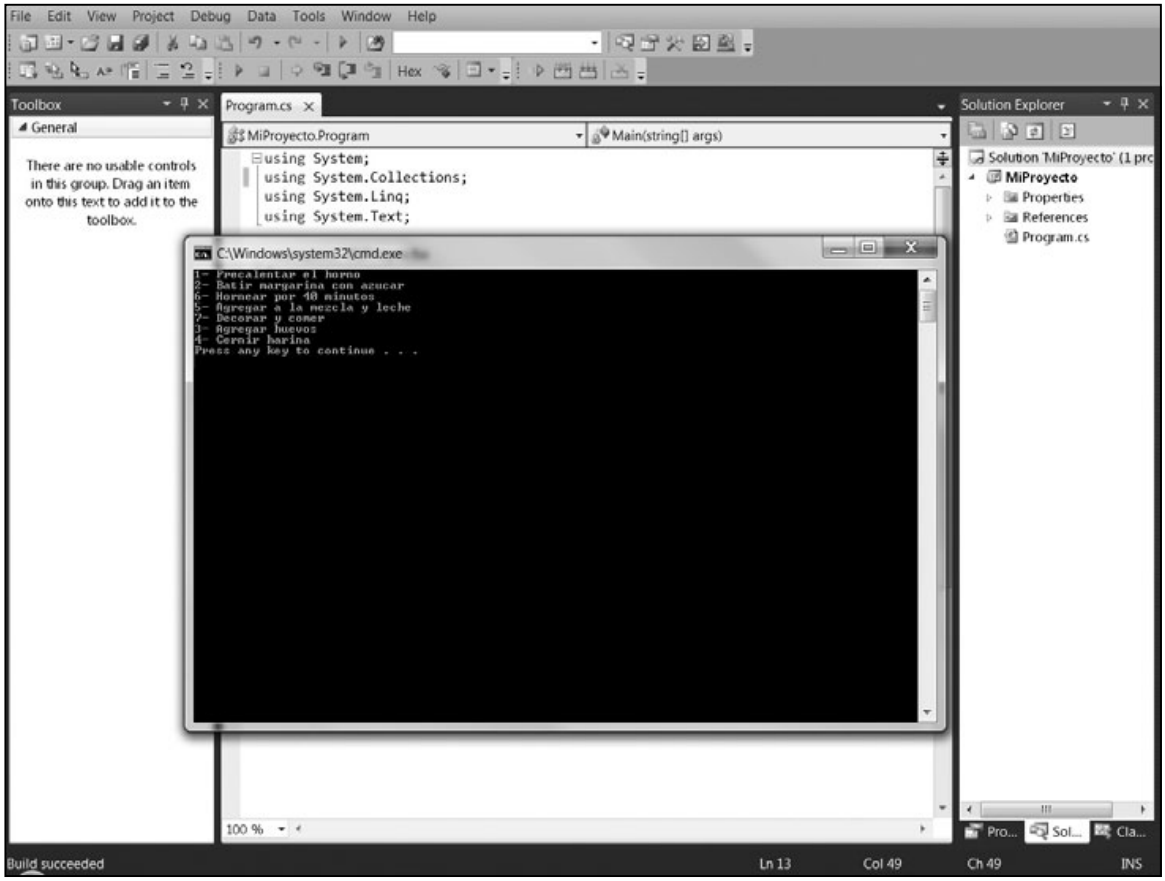

Figura 2. *Aquí vemos que los mensajes se colocan en el orden en que se encuentran en la función* ***main()*** *y no en el orden en que esperábamos.*

Podemos observar que cuando se ejecuta, las sentencias se ejecutan en el mismo orden como están en el código del programa, no se ejecutan como nosotros sabemos que se deben de ejecutar. Este punto es importante, ya que nos muestra que colocar las sentencias del programa en el orden de ejecución correcto es responsabilidad del programador. La computadora no puede leernos la mente y saber qué es lo que realmente deseamos, ésta sólo ejecuta fielmente las sentencias

en el orden en que las colocamos. Si en la vida real primero horneamos y luego batimos, no podemos esperar obtener un pastel o una torta. Por lo tanto, si en la computadora no ponemos las sentencias del programa en el orden adecuado, no podemos esperar tener un programa funcional.

Errores en los programas

Una parte importante de la programación es reconocer los errores que podemos tener con el fin de corregirlos y permitir que el programa trabaje correctamente. Podemos tener dos tipos de errores: los **errores de sintaxis** y los **errores de lógica**. Los errores de sintaxis son fáciles de corregir y se deben a que escribimos en el orden incorrecto las sentencias del programa. Estos errores se muestran cuando el programa se compila, y de esa forma se pueden buscar fácilmente para corregirlos. Los errores de lógica se deben a que hemos creado nuestro algoritmo en forma errónea y aunque todas las sentencias estén correctas, el programa no lleva a cabo lo que deseamos. Estos errores son más difíciles de corregir y la mejor forma de evitarlos es por medio de un análisis previo adecuado del problema a resolver. Más adelante aprenderemos cómo llevar a cabo este análisis.

Empecemos a reconocer los errores de sintaxis y los mensajes que da el compilador cuando los tenemos. El reconocer estos mensajes y cómo fueron ocasionados nos permitirá resolverlos rápidamente en el futuro. Para esto empezaremos a hacer unos experimentos. En el código de nuestro programa, en cualquiera de las sentencias, eliminamos el punto y coma que se encuentra al final, y luego lo compilamos para ver qué sucede. Debido a que hemos eliminado el punto y coma, tenemos un error de sintaxis y el compilador nos dice que el programa no se ha compilado exitosamente.
Dentro de nuestro compilador se mostrará la ventana de lista de errores. En esta ventana se colocan los errores encontrados por el compilador. Siempre es conveniente solucionar el problema que se encuentra en la parte superior de la lista y no el último, debido a que muchas veces, solucionando el primero, quedan solucionados otros que arrastraban su problema. El mensaje que vemos nos indica que se esperaba punto y coma, tal y como se ve en la **figura 3.**

Algunas veces sucede que olvidamos colocar la cadena entre comillas o que solamente se ha colocado la comilla del inicio. Es importante no olvidar colocar las cadenas siempre entre las dos comillas. Si no lo hacemos de esta forma, el compilador no puede encontrar correctamente la cadena o dónde finaliza. La sentencia nos marcará un error de compilación.

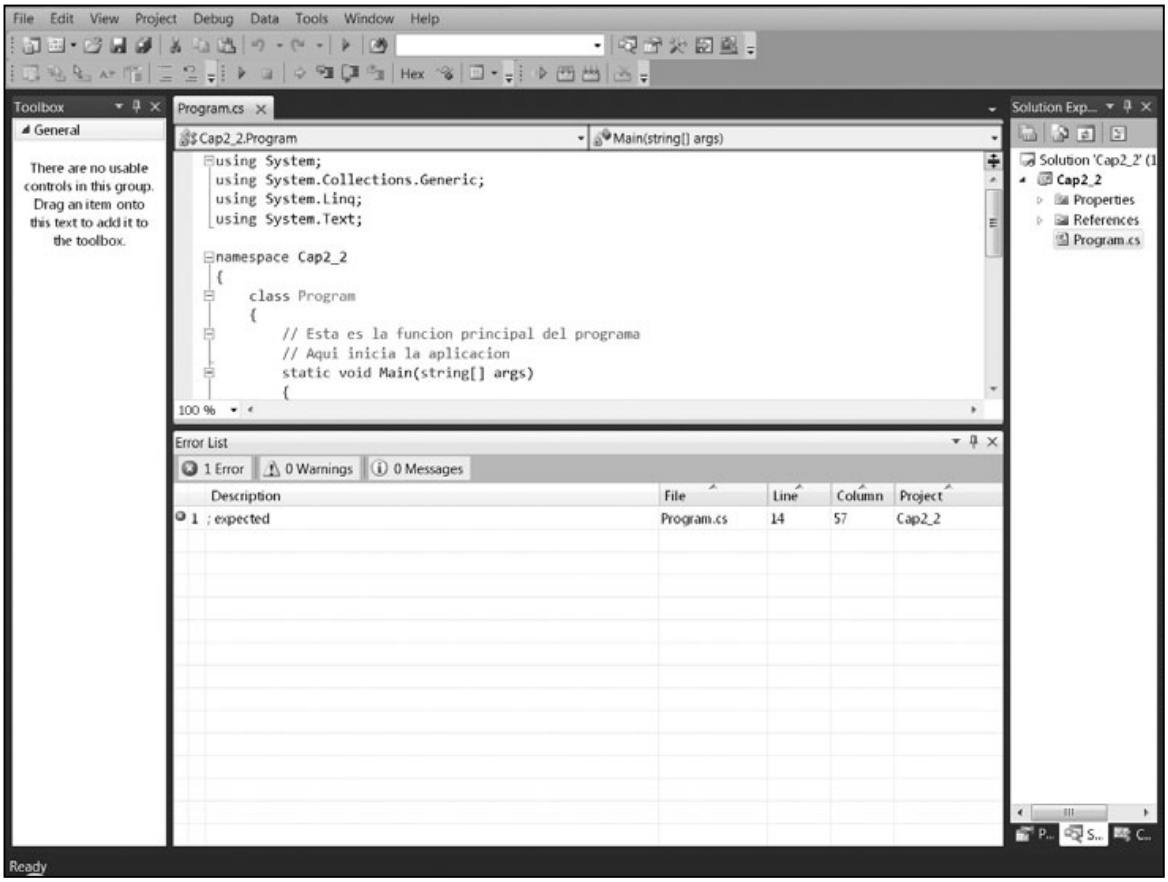

Figura 3. *Podemos ver la ventana con la lista de errores y el error de sintaxis encontrado por el compilador.*

La lista de errores nos da una descripción del error y también nos permite ver el número de línea donde se encuentra el error. Una forma sencilla de llegar a éste es simplemente hacer doble clic sobre la descripción del error, así el entorno C# nos llevará directamente a dónde se encuentra éste. Coloquemos nuevamente el punto y coma en el lugar donde estaba en un principio y compilemos. En esta ocasión, el programa puede compilar y ejecutar sin problemas, en base a la correción que hemos realizado con antelación. El error se encuentra solucionado. La solución que nos brinda C# para corregir errores es práctica y sencilla de implementar.
Ahora experimentemos otro error. A la primera sentencia le borramos el paréntesis de cierre y luego compilamos para ver qué ocurre:

Un error común que los principiantes hacen cuando nombran sus variables es que olvidan cómo nombraron a la variable y la escriben a veces con minúscula y a veces con mayúscula. Esto hace que se produzcan errores de lógica o de sintaxis. Para evitarlo, podemos utilizar una tabla que contenga la información de las variables de nuestro programa.

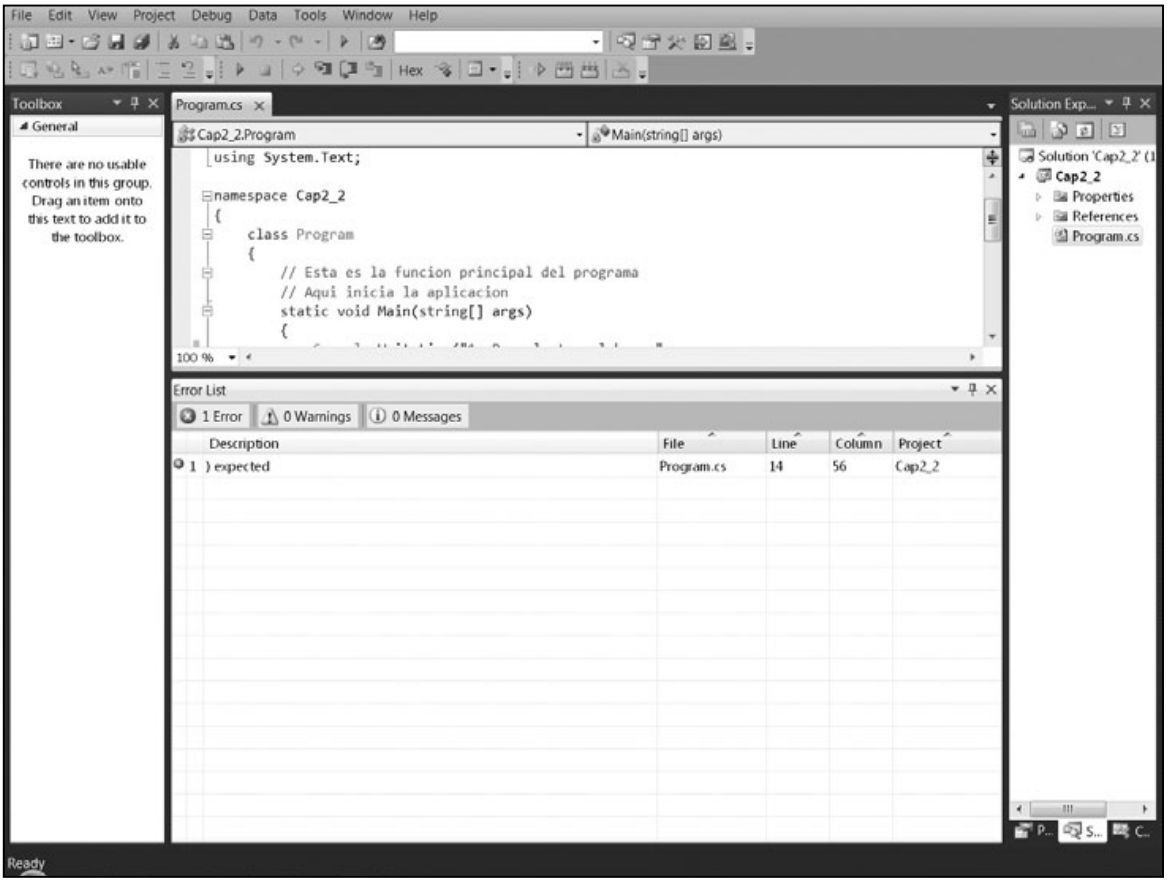

Figura 4. *Aquí vemos el mensaje de error generado por no cerrar el paréntesis.*

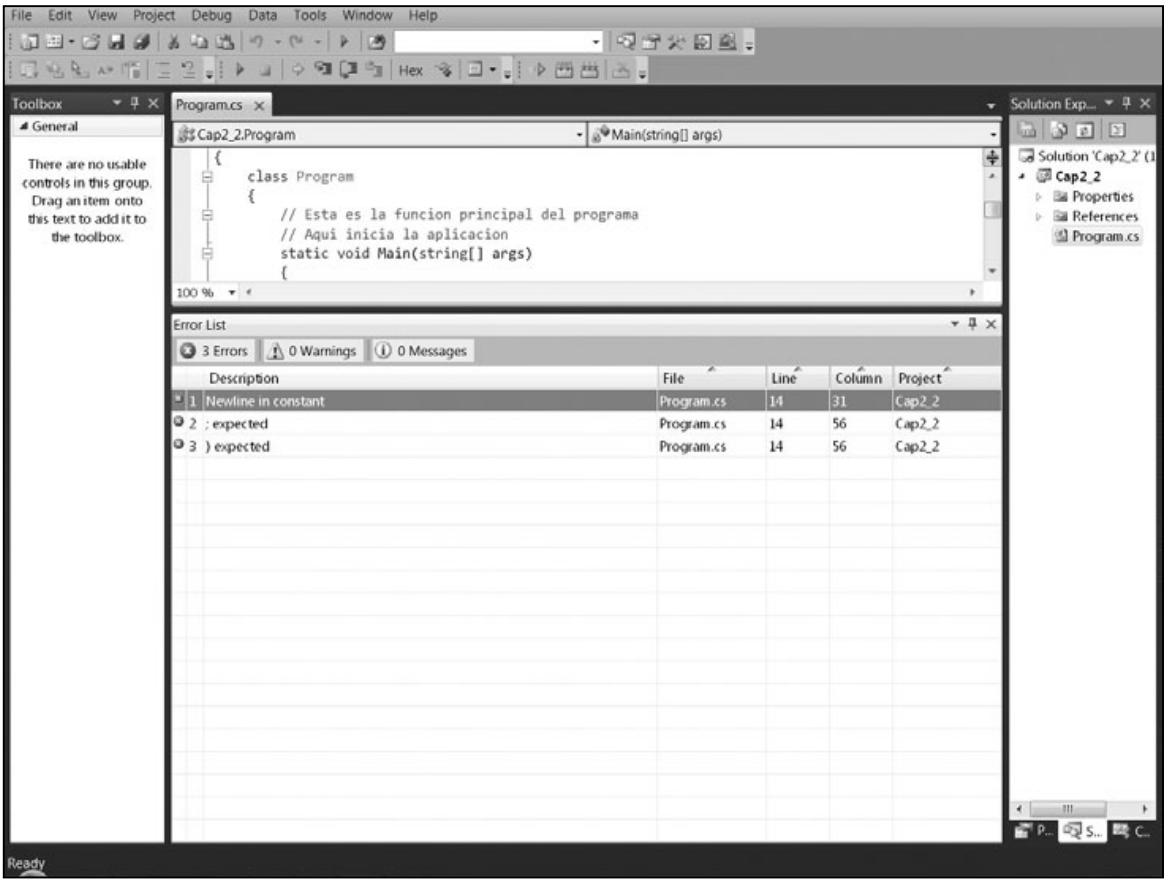

Figura 5. *Ahora nuestra lista de errores muestra tres errores en el programa.*

Corrijamos el error y compilemos de nuevo. Todo debe de estar bien. Pero lo que haremos ahora es borrar las comillas de cierre de la cadena en la primera sentencia. Este error nos presenta un escenario más interesante y nos conviene entender qué es lo que sucede. Compilemos el programa y veamos nuestra lista de errores. Ésta será mucho más extensa que las anteriores:

Aquí ha sucedido algo interesante, a pesar de que solamente hemos creado un error: en la lista de errores aparecen tres errores. Esto se debe a que un error puede ocasionar errores extras en el código siguiente. Al no cerrar la cadena, el compilador no sabe dónde termina la sentencia y cree que el paréntesis de cierre y el punto y coma forman parte de la cadena. Por eso también nos muestra que faltan el paréntesis y el punto y coma.

Debido a que los errores pueden ser en cadena, como en este ejemplo, siempre necesitamos corregir el error que se encuentra en primer lugar. Si nosotros intentáramos corregir primero el error del punto y coma, no resultaría evidente ver cuál es el error en esa línea, ya que veríamos que el punto y coma sí está escrito.

A veces sucede que nuestro programa muestra muchos errores, pero cuando corregimos el primer error, la gran mayoría desaparece. Esto se debe a factores similares a los que acabamos de ver.

La diferencia entre los métodos Write() y WriteLine()

Hemos visto que el método **WriteLine()** nos permite mostrar un mensaje en la consola, pero existe un método similar que se llama **Write()**. La diferencia es muy sencilla. Después de escribir el mensaje **WriteLine()** inserta un salto de línea, por lo que lo próximo que se escriba aparecerá en el renglón siguiente. Por su parte, el método **Write()** no lleva a cabo ningún salto de línea y lo siguiente que se escriba será en la misma línea. Veamos todo esto en el siguiente ejemplo:

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la funcion principal del programa
        // Aqui inicia la aplicacion
        static void Main(string[] args)
        {
            Console.Write("Mensaje 1 ");
```

```
            Console.Write("Mensaje 2 ");
            Console.WriteLine("Mensaje 3 ");
            Console.WriteLine("Mensaje 4 ");
            Console.Write("Mensaje 5 ");
        }
    }
}
```

Al ejecutarlo, vemos cómo en efecto después de usar **Write()** lo siguiente que se escribe se coloca en el mismo renglón, pero al usar **WriteLine()** lo siguiente aparece en el renglón siguiente. Esto es un ejemplo claro de lo explicado anteriormente:

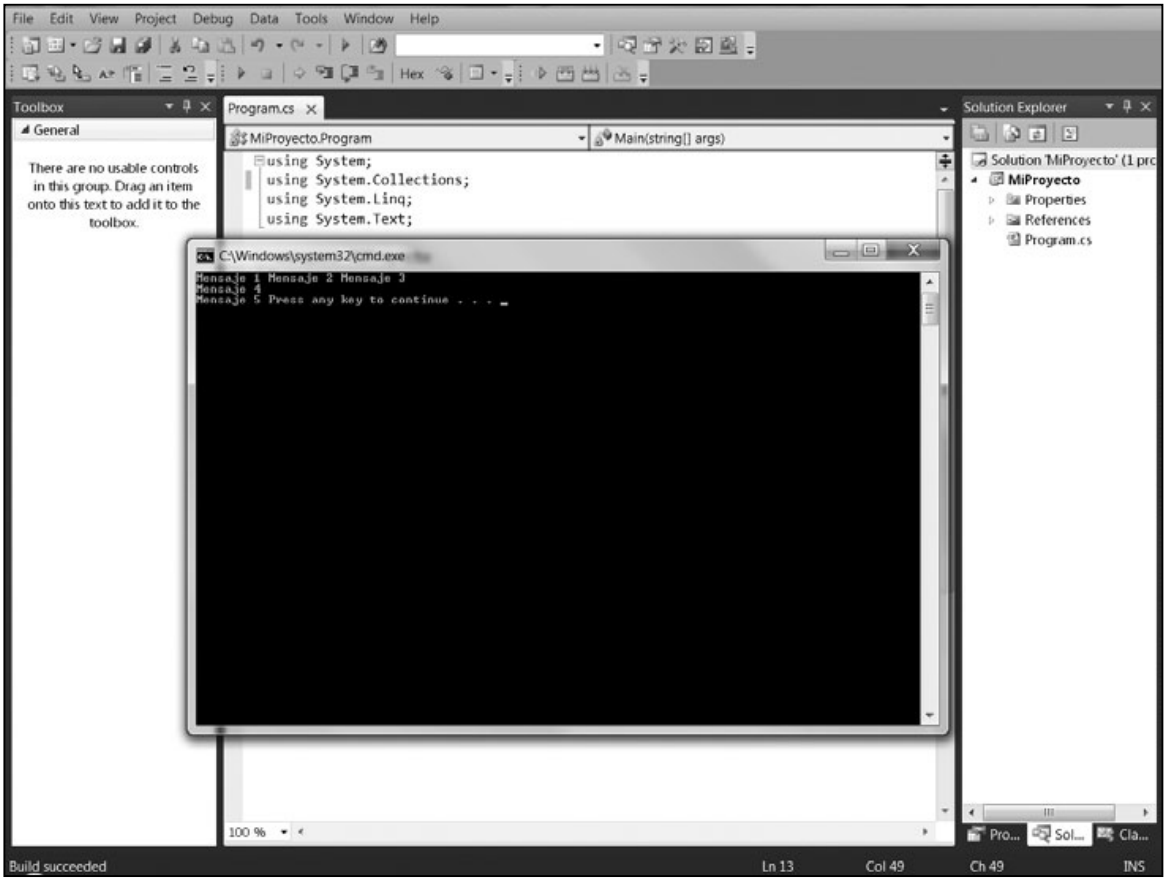

Figura 6. *Podemos observar la diferencia entre los métodos Write() y WriteLine().*

Las variables

Las computadoras no solamente escriben mensajes en la pantalla, sino que deben hacer cosas útiles y para poder hacerlas necesitan información. La información debe almacenarse y manipularse de alguna forma.

Cuando queremos guardar algo en la vida cotidiana podemos utilizar cajas, por lo que cualquier cosa que deseamos almacenar puede ser colocada en una caja de

cartón, por ejemplo. Sin embargo, el tamaño de la caja dependerá de lo que deseamos almacenar. Unos zapatos pueden caber en una caja pequeña, pero en esa caja no podemos colocar un televisor. Si empezamos a almacenar muchas cosas en cajas, pronto tendremos demasiadas y será difícil poder encontrar lo que hemos guardado. Entonces, una forma de solucionar esto es por medio de etiquetas. A cada caja le ponemos una etiqueta con el contenido y de esta forma podemos encontrar rápidamente lo que buscamos.

Las **variables** en la computadora funcionan de manera similar a nuestras cajas de almacenaje. Las podemos imaginar como pequeñas cajas que existen en la memoria de la computadora y su tamaño dependerá de la información que deben guardar. Esto se conoce como tipo de la variable. Para poder acceder a esa caja o variable le ponemos un nombre que sería la etiqueta con la que la identificamos.

Para hacer uso de una variable, lo primero que tenemos que hacer es **declararla.** La declaración de éstas es algo muy sencillo. Como primer paso tenemos que colocar el **tipo** y luego el **nombre**. Las variables en C# deben nombrarse de acuerdo con unas recomendaciones sencillas:

- Los nombres de las variables deben empezar con letra.
- Es posible colocar números en los nombres de las variables, pero no empezar el nombre con ellos.
- Los nombres de las variables no pueden llevar signos a excepción del guión bajo _ .
- Las variables no pueden llevar acentos en sus nombres.

Cuando nombramos las variables, hay que tener en cuenta que C# es **sensible** a las **mayúsculas** y **minúsculas**, por lo que una variable llamada **costo** no es la misma que otra variable llamada **COSTO** u otra llamada **Costo.**

Es recomendable nombrar a las variables con nombres que hagan referencia a la información que guardarán. Si nos acostumbramos a hacer esto desde que empezamos a programar, evitaremos muchos problemas en el futuro y será mucho más sencillo corregir nuestros programas cuando tengan errores.

Veamos un ejemplo de cómo podemos declarar una variable que guarde valores **numéricos enteros**. El tipo que guarda estos valores se conoce como **int.**

Es importante conocer los tipos y la información que pueden guardar, ya que esto nos permitirá guardar la información necesaria y utilizar la menor cantidad de memoria. Podemos aprender los rangos de los tipos o imprimir una tabla y tenerla a mano. Con la práctica podremos utilizar los tipos sin tener que verificar los rangos constantemente.

```
int costo;
```

Vemos que al final de la sentencia hemos colocado el punto y coma. Si necesitamos declarar más variables lo podemos hacer de la siguiente forma:

```
int costo;
int valor;
int impuesto;
```

Pero también es posible declarar las variables en una sola línea. Para esto simplemente separamos los nombres de las variables con comas. No hay que olvidar colocar el punto y coma al final de la sentencia.

```
int costo, valor, impuesto;
```

C# nos provee de muchos tipos para poder usar con nuestras variables, que podemos conocer en la siguiente tabla:

TIPO	INFORMACIÓN QUE GUARDA
bool	Es una variable booleana, es decir que solamente puede guardar los valores verdadero o falso (**true** o **false**) en términos de C#.
byte	Puede guardar un byte de información. Esto equivale a un valor entero positivo entre 0 y 255.
sbyte	Guarda un byte con signo de información. Podemos guardar un valor entero con signo desde –128 hasta 127.
char	Puede guardar un carácter de tipo Unicode.
decimal	Este tipo puede guardar un valor numérico con decimales. Su rango es desde ±1.0 ? 10?28 hasta ±7.9 ? 1028.
double	También nos permite guardar valores numéricos que tengan decimales. El rango aproximado es desde ±5.0 ? 10?324 hasta ±1.7 ? 10 308.

Un punto importante a tener en cuenta es que la asignación siempre se lleva a cabo de derecha a izquierda. Esto quiere decir que el valor que se encuentra a la derecha del signo igual se le asigna a la variable que se encuentra a la izquierda del signo igual. Si no lo hacemos de esta forma, el valor no podrá ser asignado o será asignado a la variable incorrecta.

TIPO	INFORMACIÓN QUE GUARDA
float	Otro tipo muy utilizado para guardar valores numéricos con decimales. Para este tipo el rango es desde ±1.5 ? 10?45 hasta ±3.4 ? 1038.
int	Cuando queremos guardar valores numéricos enteros con signo, en el rango de -2,147,483,648 hasta 2,147,483,647.
uint	Para valores numéricos enteros positivos, su rango de valores es desde 0 hasta 4,294,967,295.
long	Guarda valores numéricos enteros realmente grandes con un rango desde -9,223,372,036,854,775,808 hasta 9,223,372,036,854,775,807.
uling	Guarda valores numéricos enteros positivos. Su rango de valores varía desde 0 hasta 18,446,744,073,709,551,615.
short	Guarda valores numéricos enteros, pero su rango es menor y varía desde -32,768 hasta 32,767.
ushort	Puede guardar valores numéricos enteros positivos con un rango desde 0 hasta 65,535.
string	Este tipo nos permite guardar cadenas.

Tabla 1. *Ésta es la tabla de los tipos más utilizados en C#. Es útil para seleccionar el tipo adecuado para las variables.*

Una vez que hemos declarado la variable, debemos **inicializarla**. Inicializar una variable es asignarle un valor por primera vez. Esto nos permite darle un valor inicial que puede ser útil en la ejecución de nuestro programa (no sólo podemos asignarle a una variable un valor fijo, sino también puede ser desde un texto ingresado por el usuario o desde el registro de una base de datos).
Para asignarle un valor a una variable, ya sea durante la inicialización o durante el transcurso del programa, usamos un **operador de asignación**, el signo **igual.**
Veamos a continuación un ejemplo de cómo asignar valores:

```
costo = 50;
valor = 75;
impuesto = 10;
```

En este caso, la variable **costo** almacena el valor de 50, la variable **valor** almacena el 75, y la variable **impuesto** guarda el valor 10 .

Los comentarios solamente nos sirven a nosotros como seres humanos para facilitarnos a comprender el código del programa. El compilador ignora todos los comentarios y no los toma en cuenta en el momento de la compilación. Los comentarios no hacen el programa ni más lento ni más rápido. Sin embargo, sí son muy útiles cuando nosotros vemos el código.

Una asignación no válida sería la siguiente:

```
50 = costo;
```

En este caso nada queda asignado a la variable **costo**. Siempre la variable que recibe el valor se coloca a la izquierda del signo igual.
Si lo necesitamos, es posible inicializar la variable al momento de declararla. Esto es muy cómodo y si lo hacemos así evitamos olvidar la inicialización de la variable. Para hacerlo colocamos el tipo de la variable seguido del nombre a usar en la variable e inmediatamente con el operador de asignación colocamos el valor.

```
int costo = 50;
```

Comentarios en el programa

Los programas de computadora pueden ser muy grandes, por lo que es necesario colocar **comentarios** de codigo en el programa. Los comentarios sirven para documentar las partes del programa, explicar lo que hace cierta parte del código o simplemente colocar un recordatorio.
Podemos hacer comentarios de una sola línea, que se colocan por medio de la doble barra // seguido del texto del comentario. El texto que se encuentre a partir de la doble barra // y hasta el fin del renglón es ignorado por el compilador. Algunos ejemplos de comentarios pueden ser:

```
// Declaro mis variables
int ancho, alto;

int tarea; // Esta variable guarda el área calculada.

// La siguiente sentencia nunca se ejecuta
// Console.WriteLine("Hola");
```

Si llegáramos a necesitar colocar mucha información en nuestros comentarios, puede ser útil explayarla en varios renglones, es decir, hacer un **bloque de comentarios**. Para esto, abrimos nuestro bloque de comentarios con una barra seguida de un asterisco, /*, y finalizamos éste con un asterisco en primer lugar, seguido de una barra */. Todo lo que se encuentre entre estos delimitadores será tomado como un comentario y el compilador lo pasará por alto.
Veamos un ejemplo de un comentario de múltiples líneas:

```
/* La siguiente sección calcula el área y usa las variables
alto – para guardar la altura del rectángulo
ancho – para guardar la base del rectángulo
*/
```

Así vemos cómo los comentarios colocados en varios renglones, se toman como un solo comentario. Esto es algo muy útil para recordatorios futuros.

Mostrar los valores de las variables en la consola

Ya que podemos guardar valores, también nos gustaría poder mostrarlos cuando sea necesario. Afortunadamente, por medio del método **WriteLine()** o el método **Write()** podemos hacerlo. En este caso no usaremos la cadena como antes, lo que usaremos es una **cadena de formato** y una **lista de variables**. La cadena de formato nos permite colocar el mensaje a mostrar e indicar qué variable usaremos para mostrar su contenido y también en qué parte de la cadena queremos que se muestre. En la lista de variables simplemente colocamos las variables que queremos mostrar.
La cadena de formato se trata como una cadena normal, pero agregamos **{}** donde deseamos que se coloque el valor de una variable. Adentro de **{}** debemos colocar el **índice** de la variable en nuestra lista. Los índices inician en **cero**. Si queremos que se muestre el valor de la primera variable en la lista colocamos **{0}**, si es la segunda entonces colocamos **{1}** y así sucesivamente. La lista de variables simplemente se coloca después de la cadena, separando con comas las diferentes variables.

Por ejemplo, podemos mostrar el valor de una variable de la siguiente forma:

```
Console.WriteLine("Se tiene {0} en la variable", costo);
```

Como solamente tenemos una variable en nuestra lista de variables entonces colocamos **{0}**. Pero también podemos mostrar el valor de dos variables o más, como lo muestra el siguiente ejemplo:

```
Console.WriteLine("La primera es {0} y la segunda es {1}", costo, valor);
```

Ahora veamos un ejemplo con lo que hemos aprendido.

```
using System;
using System.Collections.Generic;
```

```
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {

            // Declaramos variables
            int costo;
            int valor, precio;

            // Inicializamos las variables
            costo = 50;
            valor = 75;
            precio = 125;

            // Declaramos e inicializamos
            int impuesto = 10;

            // Mostramos un valor
           Console.WriteLine("El valor adentro de costo es {0}", costo);

            // Varias variables
            Console.WriteLine("Valor es {0} y precio es {1}", valor,
               precio);

            // Dos veces la misma variable
           Console.WriteLine("Valor es {0} y precio es {1} con valor de
               {0}", valor, precio);

            /* No olvidemos
              mostrar el valor
              de la variable impuesto
              */
            Console.WriteLine("Y el valor que nos faltaba mostrar
               {0}",impuesto);
```

```
            }
        }
    }
```

Una vez que hemos compilado la aplicación, obtendremos el resultado que mostramos en la figura a continuación.

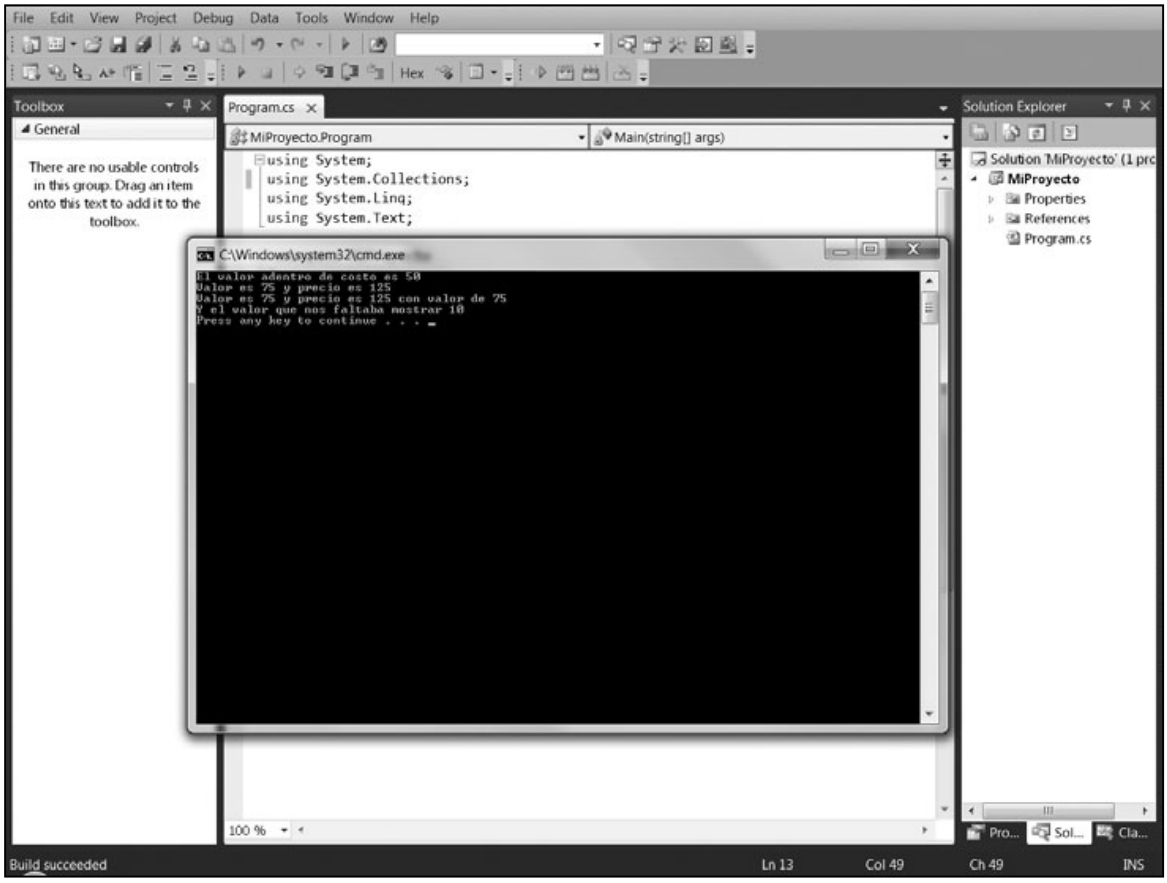

Figura 7. *En la consola observamos que los valores guardados en las variables fueron desplegados de acuerdo con lo colocado en la cadena de formato.*

Operaciones aritméticas

Poder guardar la información es muy importante, pero es mucho más importante que podamos recuperar y manipular esa información para hacer algo útil con ella. Para esto, debemos comenzar a realizar las operaciones aritméticas básicas: suma, resta, multiplicación y división, algo muy común en cualquier sistema informático.

Cuando queremos o necesitamos realizar una operación aritmética debemos hacer uso de un **operador**. Cada operación tiene su propio operador, que es el encargado de procesar los números, realizando el determinado cálculo, para luego devolvernos el resultado deseado.

OPERADOR	DESCRIPCIÓN
=	Asignación. Este operador ya es conocido por nosotros.
+	Suma. Nos permite sumar los valores de las variables o los números
-	Resta. Para restar los valores de las variables o los números.
*	Multiplicación. Multiplica los valores de las variables o los números.
/	División. Divide los valores de las variables o los números.
%	Módulo. Nos da el residuo de la división.

***Tabla 2.** Ésta es la tabla de operadores aritméticos en C#.*

Veamos algunos ejemplos de cómo utilizar estos operadores. Imaginemos que ya hemos declarado e inicializado las variables. Para guardar en la variable **resultado** la suma de dos números, hacemos lo siguiente:

```
resultado = 5 + 3;
```

Como hemos visto, la expresión a la derecha se le asigna a la variable, por lo que en este caso la expresión **5 + 3** se evalúa y su resultado que es **8** es almacenado en la variable **resultado**. Al finalizar esta sentencia, la variable **resultado** tiene un valor de **8**, que podrá ser utilizado dentro de la misma función o en el procedimiento donde fue declarado.
Las operaciones también se pueden realizar con las variables y los números. Supongamos que la variable **a** ha sido declarada y se le ha asignado el valor de **7.**

```
resultado = a – 3;
```

De nuevo se evalúa primero la expresión **a-3**, como el valor adentro de **a** es **7**, entonces la evaluación da el valor de **4**, que se le asigna a la variable **resultado.**
Si queremos podemos trabajar únicamente con variables. Ahora supondremos que la variable **b** ha sido declarada y que le hemos asignado el valor **8.**

```
resultado = a * b;
```

Entonces, se evalúa la expresión **a*b** que da el valor **56**, que queda asignado a la variable **resultado** al finalizar la expresión.
Si lo que deseamos es la división, podemos hacerla de la siguiente forma:

```
resultado = a / b;
```

En este caso, se divide el valor de **a** por el valor de **b**. El resultado es **0,875**. Como el número tiene valores decimales, debemos usar variables de algún tipo que nos permita guardarlos como **float** o **double.**

Y por último nos queda el módulo.

```
resultado = a % b;
```

En este caso recibimos en resultado el valor del residuo de dividir **a** por **b.**

Veamos un pequeño programa donde se muestra lo aprendido en esta sección.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Declaramos las variables, ahora de tipo flotante
            float a, b, resultado;

            // Inicializamos las variables
            a = 7;
            b = 8;
```

Los errores más comunes en las expresiones matemáticas son olvidar el punto y coma, hacer la asignación errónea, escribir mal el nombre de la variable y agrupar incorrectamente las operaciones. Utilizar los paréntesis para agrupar ayuda mucho a reducir errores.

```
resultado = 0;

    // Sumas
    Console.WriteLine("Sumas");

    resultado = 3 + 5;
    Console.WriteLine("Resultado = {0}",resultado);

    resultado = a + 3;
    Console.WriteLine("Resultado = {0}", resultado);

    resultado = a + b;
    Console.WriteLine("Resultado = {0}", resultado);

    // Restas
    Console.WriteLine("Restas");

    resultado = a - b;
    Console.WriteLine("Resultado = {0}", resultado);

    resultado = b - 5;
    Console.WriteLine("Resultado = {0}", resultado);

    resultado = b - a;  // A la variable b se le resta a
    Console.WriteLine("Resultado = {0}", resultado);

    // Multiplicaciones
    Console.WriteLine("Multiplicaciones");

    resultado = a * 5;
    Console.WriteLine("Resultado = {0}", resultado);

    resultado = a * 3.5f;
    Console.WriteLine("Resultado = {0}", resultado);

    resultado = a * b;
    Console.WriteLine("Resultado = {0}", resultado);

    // Divisiones
    Console.WriteLine("Divisiones");
```

```
            resultado = a / 3;
            Console.WriteLine("Resultado = {0}", resultado);

            resultado = a / b;
            Console.WriteLine("Resultado = {0}", resultado);

            resultado = b / 2.5f;
            Console.WriteLine("Resultado = {0}", resultado);

            // Modulo
            Console.WriteLine("Modulo");

            resultado = a % b;
            Console.WriteLine("Resultado = {0}", resultado);
        }
    }
}
```

Podemos ver la ejecución en la siguiente figura:

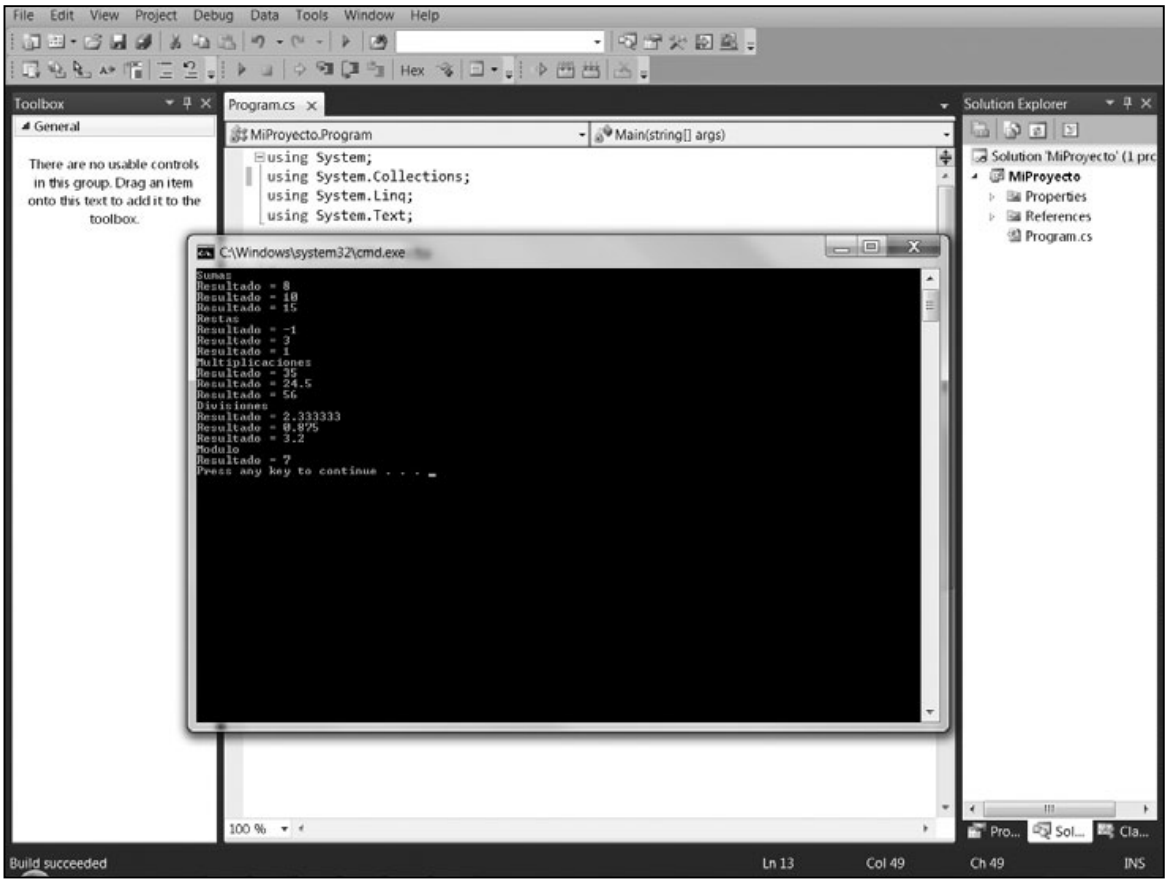

Figura 8. *Vemos los resultados de las operaciones aritméticas sobre nuestras variables.*

Precedencia de operadores

Hasta el momento hemos utilizado expresiones aritméticas sencillas, pero en la vida real se pueden necesitar expresiones más complicadas. Entonces, si estas expresiones no se colocan adecuadamente, esto nos puede ocasionar problemas.

Por ejemplo, veamos la siguiente expresión.

```
resultado = 3 * 5 + 2;
```

Esta expresión presenta un problema para nosotros en este momento. No sabemos si primero hace 3*5 y le suma 2, que da como resultado 17, o si suma 5+2 y luego lo multiplica por 3, lo que da como resultado 21. ¿Cuál es la respuesta correcta?
La forma de saberlo es por medio de la **precedencia de operadores**. Ésta nos indica el orden en el que se llevan a cabo las operaciones. Este orden depende del tipo de operador que se utiliza. Algunos operadores tienen más precedencia que otros, es decir, se ejecutan primero.
La siguiente tabla muestra la precedencia de los operadores que hemos visto hasta el momento en C#. Se encuentran listados de mayor a menor precedencia.

OPERADOR	DESCRIPCIÓN
*	Multiplicación
/	División
%	Módulo
+	Adición
-	Sustracción

***Tabla 3.** Esta tabla nos muestra la precedencia de operadores en C#. La multiplicación tiene más precedencia y se ejecuta primero.*

Esto quiere decir que cuando tenemos una expresión aritmética como la del ejemplo, primero se llevaría a cabo la multiplicación y luego la suma, entonces con esto podemos deducir cómo C# evaluaría la expresión.
Pero algunas veces sería mejor si nosotros pudiéramos organizar nuestra expresión con el fin de hacerla más fácil de leer para nosotros o para que podamos indicar de forma precisa cómo hacer la operación. Para que podamos organizar una expresión hacemos uso de los paréntesis. Cada sección que tengamos en los paréntesis se evalúa como una expresión y el resultado se pasa al resto de la expresión. Supongamos que tenemos la siguiente expresión:

```
resultado = (3*5) + 2;
```

Vemos que por medio de los paréntesis se ha agrupado una parte de la expresión, por lo que se evalúa y equivale a:

```
resultado = 15 + 2;
```

Lo que al final evalúa en:

```
resultado = 17;
```

Veamos otro ejemplo de cómo podemos usar los paréntesis para indicar cómo deseamos que se evalúe la expresión.

```
resultado = 3 * (5+2);
```

Lo primero que sucede es que se evalúa la expresión contenida en el paréntesis.

```
resultado = 3 * 7;
```

Que nos da el valor:

```
resultado = 21;
```

Veamos un ejemplo un poco más complicado.

```
resultado = (3+7) * (36 + 4 *(2+5));
```

Veamos qué sucede:

```
resultado = 10 * (36 + 4 * 7);
```

Luego se continúa evaluando:

```
resultado = 10 * (36 + 28);
```

Lo que da como resultado:

```
resultado = 10 * 64;
```

Y al final obtenemos:

```
resultado = 640;
```

Afortunadamente, C# hace todas estas evaluaciones y todos estos cálculos por nosotros, pero ahora sabemos cómo sucede y sabemos que podemos usar los paréntesis para organizar nuestras expresiones aritméticas.

Cómo pedirle datos al usuario

Hemos utilizado variables para guardar valores y con los operadores aritméticos hemos podido realizar cálculos matemáticos. Sin embargo, todos los valores que hemos usado se encuentran colocados directamente en el código del programa. Esto no es muy cómodo, ya que si deseamos hacer un cálculo con diferentes valores, es necesario modificar el código y compilar nuevamente. Sería más útil si pudiéramos hacer que el usuario colocara los valores que necesita cuando el programa se ejecuta. Para esto, C# nos provee de un método que pertenece a la clase **Console**. El método se llama **ReadLine()**. Éste no necesita ningún parámetro, por lo que sus paréntesis permanecerán vacíos, y nos regresa una cadena que contiene lo que escribió el usuario con el teclado.

Ésta es una forma sencilla para pedirle información al usuario. Supongamos que deseamos pedirle al usuario su nombre, que utilizaremos a posteriori para saludarlo. Para eso colocamos el siguiente código:

```
string entrada = " ";
```

Un error común con expresiones grandes es olvidar cerrar un paréntesis en forma adecuada. En Visual Studio es fácil ver cuáles paréntesis son pareja ya que al seleccionar uno de ellos su pareja se ilumina. Para cada paréntesis de apertura forzosamente debe de existir un paréntesis de cierre. Cuando esto no se logra se denomina desbalance de paréntesis.

```
Console.WriteLine("Escribe tu nombre");
entrada=Console.ReadLine();

Console.WriteLine("Hola {0}, como estas?",entrada);
```

En este fragmento de programa es importante notar cómo el valor obtenido por **ReadLine()** se le asigna a la variable de tipo cadena **entrada**. Si no colocamos una variable que recibe el valor de lo escrito por el usuario, éste se perderá. Cuando usemos **ReadLine()** debemos hacerlo como se muestra. Una vez que recibimos la entrada del usuario podemos trabajar con ella sin problemas.

Conversión de variables

El método de **ReadLine()** nos presenta una limitación en este momento ya que sólo regresa el valor como cadena, pero no podemos utilizar una cadena en operaciones aritméticas o asignarla directamente a una variable numérica.

Lo que debemos hacer es convertir ese dato numérico guardado como cadena a un valor numérico útil para nuestros propósitos. C# nos provee de una clase que permite hacer conversiones entre los diferentes tipos de variables. Esta clase se conoce como **convert** y tiene una gran cantidad de métodos para la conversión. Aquí sólo veremos cómo convertir a enteros y flotantes, ya que son los tipos con los que trabajaremos en este libro principalmente. Sin embargo, los demás métodos se usan de forma similar y podemos encontrar la información necesaria sobre éstos en **MSDN** (*MicroSoft Developer Network*).

Si queremos convertir del tipo **string** al tipo **int** usaremos el método **ToInt32()**. Éste necesita un parámetro, que es la cadena que deseamos convertir. El método regresa un valor de tipo **int**, por lo que es necesario tener una variable que lo reciba.

Un ejemplo de esto es:

```
a = Convert.ToInt32(entrada);
```

MSDN es un sitio web donde encontraremos información técnica sobre todas las clases y funciones, y todos los métodos que conforman los lenguajes que componen Visual Studio. También es posible encontrar ejemplos y descripciones de los errores de compilación. Tiene un buen sistema de búsquedas que resulta muy útil. La dirección es **www.msdn.com**.

De forma similar podemos convertir la cadena a una variable de tipo **float**. En este caso usaremos el método **ToSingle()**. Éste también necesita un parámetro, que es la cadena que contiene el valor a convertir. Como regresa un valor flotante, entonces debemos tener una variable de tipo **float** que lo reciba.

```
valor = Convert.ToSingle(entrada);
```

Ya que tenemos los valores convertidos a números, entonces podemos hacer uso de ellos. Veamos un ejemplo donde le pedimos información al usuario.

```
using System;
using System.Collections.Generic;
using System.Text;

namespace AplicacionBase
{
    class Program
    {
        // Esta es la función principal del programa
        // Aquí inicia la aplicación
        static void Main(string[] args)
        {
            // Declaramos variables
            string entrada = "";
            int a = 0, b = 0, resultado = 0;

            // Leemos una cadena
            Console.WriteLine("Escribe tu nombre");
            entrada = Console.ReadLine();

            Console.WriteLine("Hola {0}, como estas?", entrada);

            // Leemos dos valores y los sumamos.

            Console.Write("Dame un entero:");
            entrada = Console.ReadLine();

            // Convertimos la cadena a entero
            a = Convert.ToInt32(entrada);
```

```
            Console.Write("Dame otro numero entero:");
            entrada = Console.ReadLine();

            // Convertimos la cadena a entero
            b = Convert.ToInt32(entrada);

            // Sumamos los valores
            resultado = a + b;

            // Mostramos el resultado
            Console.WriteLine("El resultado es {0}", resultado);

        }
    }
}
```

Ahora compilemos y ejecutemos la aplicación. Para introducir los datos, simplemente los escribimos con el teclado y al finalizar oprimimos la tecla **ENTER** o **RETURN.**

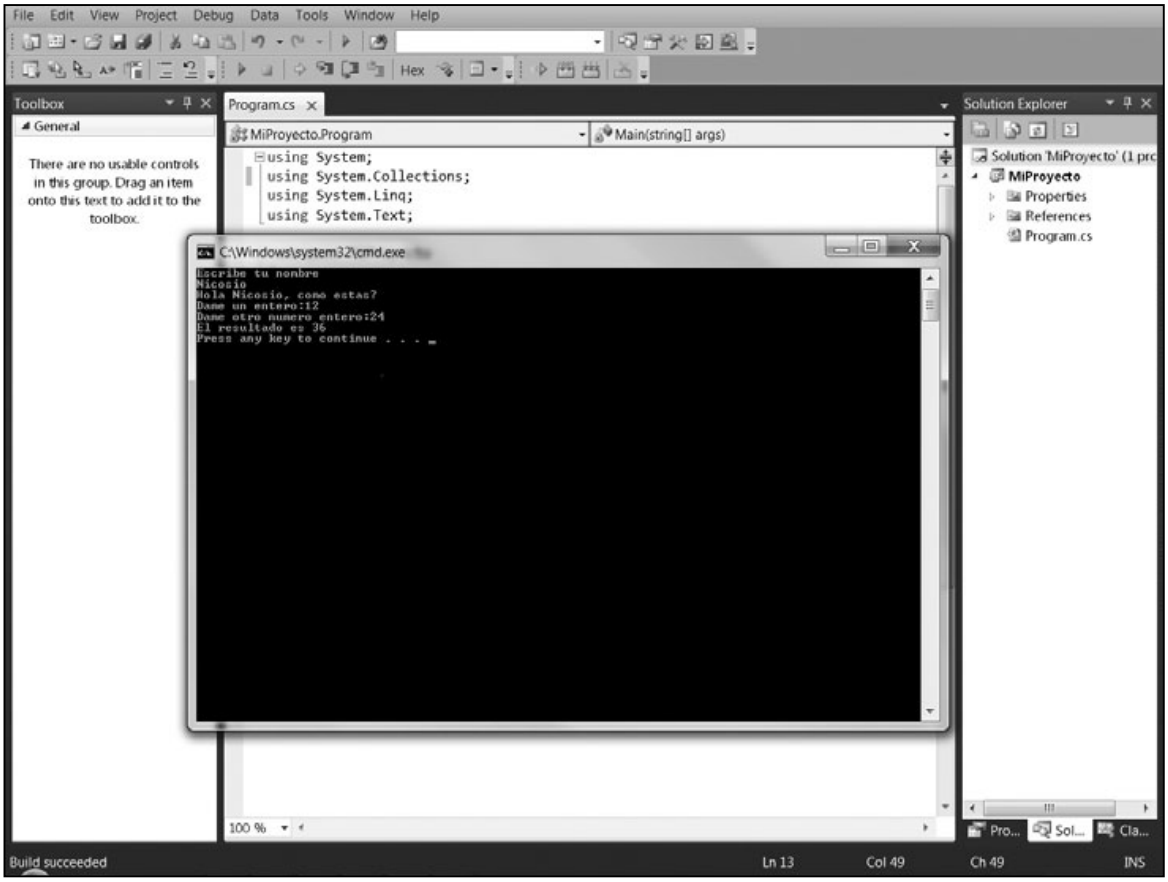

Figura 9. *Éste es el resultado de la ejecución, donde vemos que la suma se lleva a cabo con los valores introducidos por el usuario.*

Cómo resolver problemas en la computadora

Ya que conocemos algunos elementos del lenguaje, ahora es importante aprender cómo resolver los problemas en la computadora. Existen diferentes metodologías para hacerlo, pero como éste es un libro dirigido a gente sin experiencia en programación aprenderemos un método sencillo. Como ya hemos mencionado, C# es un lenguaje orientado a objetos. Sin embargo, primero aprenderemos una metodología estructurada, lo que nos permitirá tener experiencia en la resolución de problemas y luego podremos aprender la metodología orientada a objetos de manera más sencilla.

La metodología que aprendemos se basa en el concepto de subdivisión de problemas. Es decir que tomamos el problema general y luego lo dividimos en problemas más pequeños y si es necesario, dividimos estos subproblemas en problemas más pequeños, hasta que podamos resolver el subproblema directamente.

Para resolver cualquier problema, lo primero que tenemos que hacer es entenderlo. Hay que entender de forma precisa qué es lo que se pide. Los problemas tendrán información. En muchos casos, el problema nos da directamente la información. A este tipo de información se lo conoce como **información explícita**. Pero también a veces sucede que el problema no nos da directamente la información, por eso es importante entenderlo bien. En este caso nosotros debemos inferir la información de otros datos que nos proporciona o buscarla en algún otro lugar. A este tipo de información se lo conoce como **información implícita**. Afortunadamente, la información implícita en muchos casos es fácil de encontrar o forma parte de los conocimientos generales de cualquier persona.

Ya que tenemos el problema subdividido y los datos reconocidos, procedemos a realizar el algoritmo. Recordemos que el algoritmo es la serie de pasos necesarios para resolver el problema. Además, éste puede definirse por medio de un diagrama de flujo o escribirse en algo que denominaremos **pseudocódigo**. El programador puede crear el algoritmo con su método de preferencia.

Cuando se ha terminado el algoritmo, procedemos a probarlo. En esta etapa podemos probar en el papel si todo funciona bien. Si hubiera algún resultado erróneo, entonces se puede corregir el algoritmo y el proceso se repite hasta que tengamos un algoritmo que funcione adecuadamente.

Tambien es posible convertir de números a cadenas. Para esto se usa el método **ToString()**, que está disponible en todos los tipos numéricos de C#. De esta forma podemos utilizar la información de las variables numéricas en funciones o métodos que trabajen con cadenas. El método no cambia el tipo, simplemente regresa una cadena que contiene el texto de ese valor.

El último paso consiste en pasar el algoritmo a programa de computadora. Si todo se ha realizado correctamente, el programa deberá ejecutarse al primer intento. Mucha gente se sorprende que para hacer un programa de computadora, el último paso sea colocarlo en la computadora, pero ésta es la forma correcta. Algunos programadores no profesionales intentan hacer el programa directamente en la computadora, pero esto acarrea muchos errores de lógica y datos, y el tiempo en que resuelve el mismo problema es mayor y es difícil que se ejecute correctamente al primer intento. Aunque la metodología que acabamos de presentar parece larga, en realidad puede ahorrar hasta un 50% del tiempo de resolución en comparación con alguien que programe directamente en la computadora. Ahora que empezamos, es bueno adquirir hábitos correctos de programación desde el inicio. En el futuro, en especial cuando los problemas que se resuelvan sean complicados, se apreciarán los beneficios.

Elementos de un diagrama de flujo

Es momento de aprender los componentes básicos de un **diagrama de flujo**. Con estos diagramas podemos representar el algoritmo y son fáciles de hacer y entender. Otra ventaja que tienen es que son independientes del lenguaje, lo que nos da la flexibilidad de resolver el problema y usar nuestra solución en cualquier lenguaje.

El diagrama de flujo es una **representación gráfica del algoritmo** y tiene diferentes componentes. Cada componente es una figura y ésta representa un tipo de actividad. En el interior de la figura colocamos una descripción del paso que está sucediendo ahí. Por medio de flechas indicamos cuál es el paso siguiente y nos da la idea del flujo general del algoritmo. En algunas ocasiones, a las flechas les agregamos información extra, pero esto lo veremos más adelante. Aunque en este momento no utilizaremos todos los componentes del diagrama de flujo, sí conoceremos los más importantes, ya que en los capítulos posteriores los utilizaremos.

El primer componente tiene forma de ovalo y se lo conoce como **terminador**. Éste nos sirve para indicar el inicio o el final del algoritmo. Al verlo podemos saber inmediatamente dónde inicia el algoritmo y poder continuar a partir de ahí. Hay que recordar que los algoritmos deben tener un final para que sean válidos. Si el terminador es de inicio, colocamos en su interior el mensaje: **Inicio**. Si el terminador indica dónde se acaba el algoritmo, entonces se coloca el mensaje: **Final.**

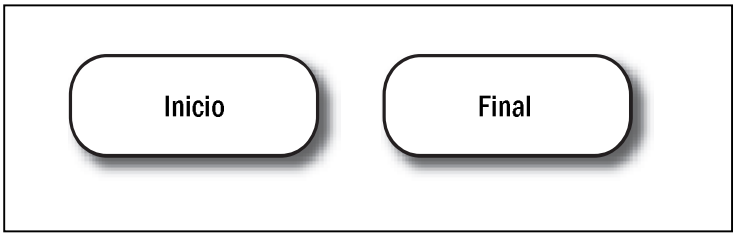

Figura 10. *En esta figura podemos observar el terminador tanto en su papel como punto de inicio como punto final del algoritmo.*

El siguiente elemento se conoce como **proceso**. Éste tiene la forma de un rectángulo y nos sirve para indicar que precisamente un proceso debe llevarse a cabo en ese lugar. La descripción del proceso se indica en su interior.

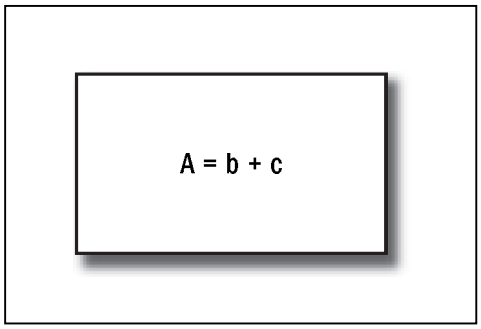

Figura 11. *El proceso se indica por medio de un rectángulo. En su interior indicamos lo que se tiene que procesar.*

Similar al proceso, tenemos el **proceso predefinido**. Su forma también es un rectángulo, pero éste tiene dos franjas a los lados. Se usa para indicar que haremos uso de un proceso que ya ha sido definido en otra parte. Un ejemplo de esto sería un método, como los que ya hemos usado. Principalmente lo usaremos para indicar llamadas a funciones o métodos definidos por nosotros mismos. En un capítulo posterior aprenderemos cómo hacer nuestros propios métodos.

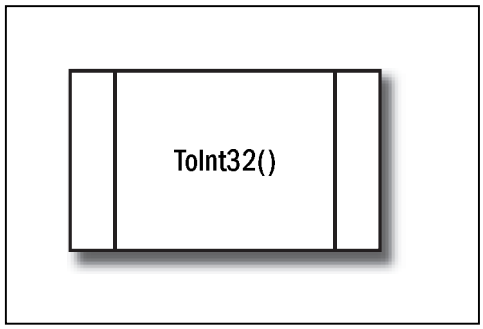

Figura 12. *Éste es el símbolo usado para el proceso predefinido.*

Nuestro próximo elemento es conocido como **condición**. Utilizaremos este elemento cuando aprendamos cómo hacer que el programa tome sus propias

III

Algunas veces necesitaremos variables que nos apoyen para resolver algún cálculo o proceso. A éstas las llamaremos datos de trabajo. Los datos de trabajo se descubren cuando hacemos el análisis del problema. Si llegamos a omitir alguno durante el análisis siempre es posible agregarlo durante el desarrollo, pero esto debe ser la excepción a la regla.

decisiones. Básicamente, sirve para hacer una selección entre las posibles rutas de ejecución del algoritmo, que dependerá del valor de la expresión evaluada. Su forma es la de un rombo y a partir de las esquinas de éste surgirán las flechas hacia las posibles rutas de ejecución del algoritmo.

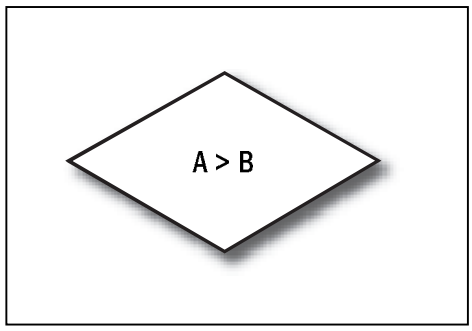

Figura 13. *La decisión se representa con un rombo.*

Luego nos encontramos con un componente conocido como **datos**. Su forma es la de un paralelogramo y podemos usarlo para indicar la salida o la entrada de los datos. Un ejemplo de esto sería cuando llevamos a cabo la petición de un valor al usuario o cuando mostramos un valor en pantalla. Adentro de él indicamos qué dato se pide o se presenta.

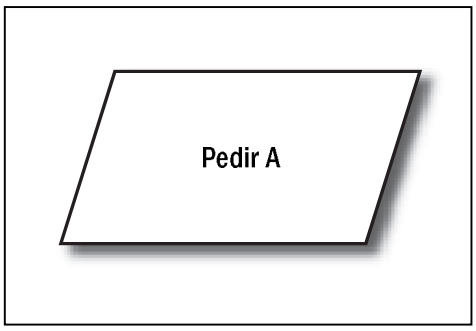

Figura 14. *En este caso usamos el dato para indicar la petición del valor de una variable.*

Hemos visto los elementos principales del diagrama de flujo. Ahora podemos hacer un ejercicio que muestre la solución de un problema en la computadora.

Es una versión gratuita de C# con las herramientas básicas para programar en este lenguaje. Esta versión nos provee todo lo necesario para los temas que aprenderemos en este libro. Aunque la versión es gratuita, es necesario registrarla para poder utilizarla por más de treinta días. La podemos descargar de **http://msdn2.microsoft.com/en-us/express/aa700756.aspx**.

Resolución de problemas en la computadora

Empecemos por un problema sencillo y fácil de entender. Para este problema seguiremos todos los pasos que hemos mencionado hasta llegar al programa de cómputo final. En primer lugar, tenemos la descripción del problema.

Hacer un programa de cómputo que calcule el área y el perímetro de un rectángulo dados sus lados. Una vez que tenemos el problema y lo hemos entendido, entonces procedemos a subdividirlo en problemas más pequeños que sean fáciles de resolver. Esta subdivisión puede ser hecha por medio de un diagrama y para esto hay que recordar que si alguno de los problemas aún continua siendo difícil, lo podemos seguir subdividiendo. A modo de ejemplo exageraremos un poco la subdivisión en este caso. El problema quedaría subdividido como se muestra en la siguiente figura.

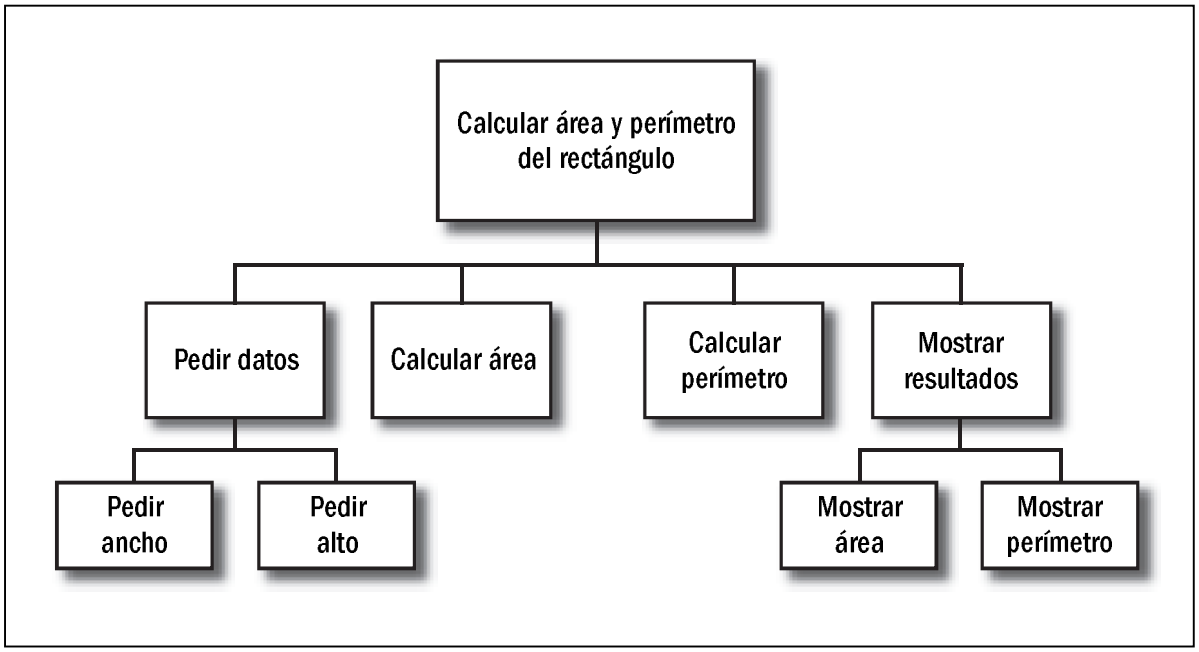

Figura 15. *Aquí tenemos nuestro problema subdividido en problemas menores y fáciles de resolver.*

Ya que conocemos el problema y lo hemos subdividido en varios subproblemas, podemos identificar de manera más clara los datos que nos hacen falta para poder trabajar en él. La descripción del problema nos proporciona dos datos explícitos: el

Si queremos encontrar información sobre cualquier clase o método de C#, el lugar apropiado para hacerlo es el sitio web MSDN que nos provee Microsoft. Cuando realicemos la búsqueda debemos filtrar para buscar únicamente contenidos del lenguaje C# y así evitar confusiones con información de otros lenguajes. La dirección es **www.msdn.com**.

área y el **perímetro**. Pero también reconocemos inmediatamente dos datos implícitos que son necesarios: el **alto** y el **ancho** del rectángulo.
Nosotros sabemos que necesitamos una cadena para recibir el valor escrito por el usuario, pero como tal, esta cadena no forma parte del problema. Sin embargo, nos ayuda a resolverlo, por lo que la cadena será un dato de trabajo. Los datos que hemos identificados serán colocados en una tabla de la siguiente forma:

NOMBRE	TIPO	VALOR INICIAL
área	Float	0.0
perímetro	Float	0.0
ancho	Float	1.0
alto	Flota	1.0
valor	String	""

Tabla 4. *Tabla de variables para el programa que estamos resolviendo.*

Hemos decidido usar el tipo **float**, ya que podemos tener valores decimales en el cálculo del perímetro. Como un rectángulo con ancho o alto de 0 no tiene sentido, hemos seleccionado como valor inicial para estas variables a 1.
Ahora viene el paso más importante: la creación del algoritmo. Sin embargo, si observamos la forma como subdividimos el problema, podemos ver que nos da una indicación de los pasos que son necesarios colocar en el algoritmo.

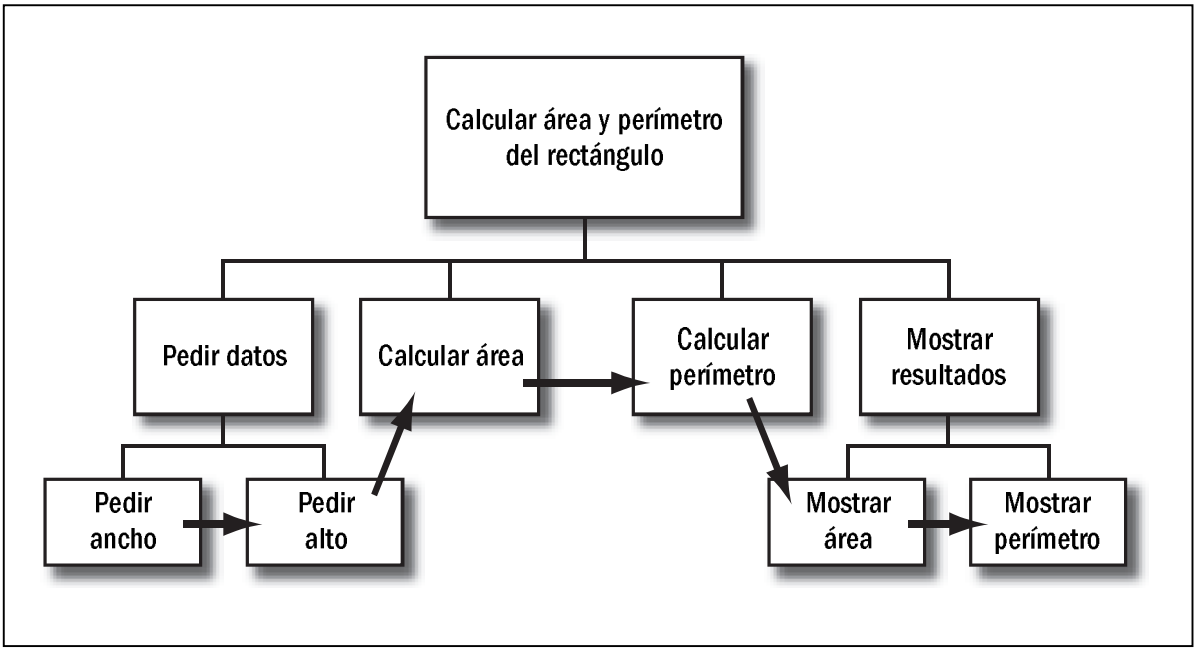

Figura 16. *La subdivisión nos muestra una lista de los pasos posibles necesarios en la subdivisión del algoritmo.*

Con esta información creamos el algoritmo con el diagrama de flujo. En éste aparece la secuencia de pasos en el orden correcto. Veamos cómo queda el diagrama.

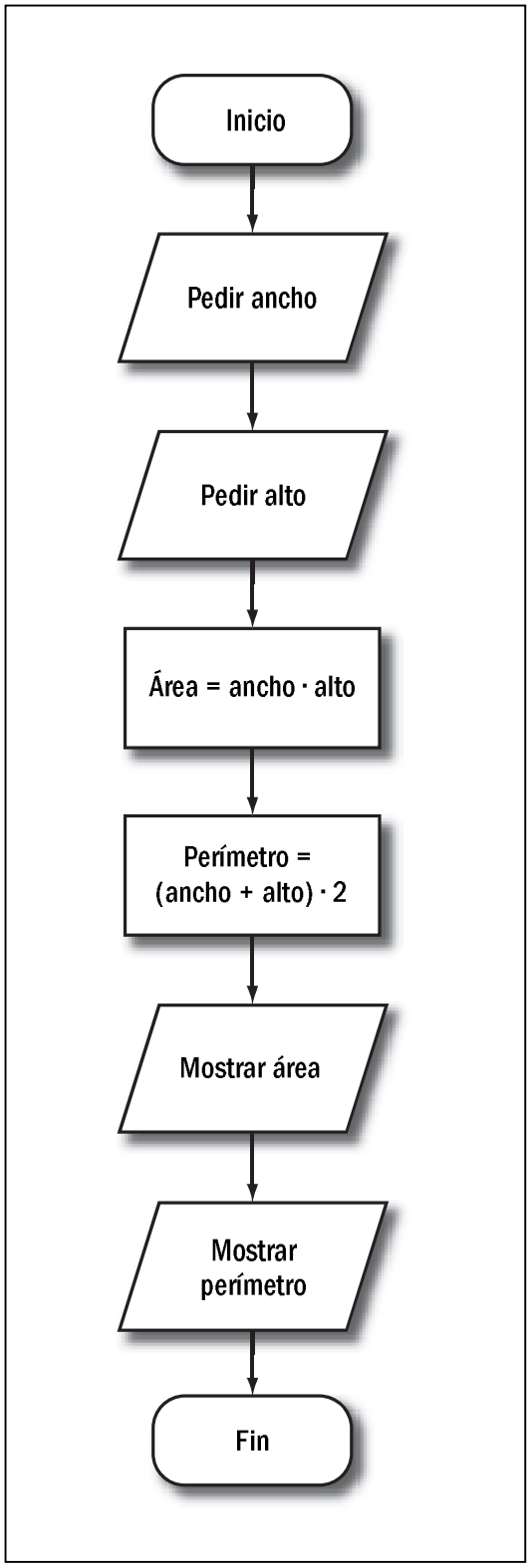

Figura 17. *El diagrama de flujo muestra el algoritmo. Las diferentes figuras nos indican el tipo de actividad que se realiza.*

Podemos observar cómo hemos utilizado el trapezoide cuando es necesario pedirle o mostrarle un dato al usuario. Los cálculos con la fórmula correspondiente están en los rectángulos, ya que son procesos, y el inicio y fin del algoritmo están representados por los terminadores.

El código de nuestra aplicación

Ya tenemos todo lo necesario para poder crear el código del programa. Para esto haremos uso de la aplicación base que ya tenemos. Como creamos una tabla de datos, ya conocemos todas las variables necesarias para la aplicación. Entonces tomamos esta tabla y colocamos los datos como variables.

```
// Declaramos las variables que necesitamos
float area = 0.0f;
float perimetro = 0.0f;
float ancho = 1.0f;
float alto = 1.0f;
string valor = "";
```

Con las variables definidas, entonces procedemos a codificar cada uno de los pasos del algoritmo. Recorremos paso por paso y en el mismo orden en que se encuentran, y vemos que lo primero es pedir el ancho del rectángulo.

```
Console.Write("Dame el ancho del rectángulo: ");
valor = Console.ReadLine();

ancho = Convert.ToSingle(valor); // Convertimos a flotante
```

Lo siguiente que hace el algoritmo es pedir el alto del rectángulo, entonces el código necesario es el siguiente.

```
Console.Write("Dame el alto del rectángulo: ");
valor = Console.ReadLine();

alto = Convert.ToSingle(valor); // Convertimos a flotante
```

Aprender a programar requiere de mucha práctica y mucha experimentación. Es bueno probar con los programas y ver qué es lo que sucede. No debemos temer en cometer errores cuando estamos aprendiendo. También es bueno verificar la documentación para las funciones, las clases y los métodos ya que es posible encontrar utilidades en éstas para el futuro.